# Software Engineering I (02161)

## Week 6

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2016

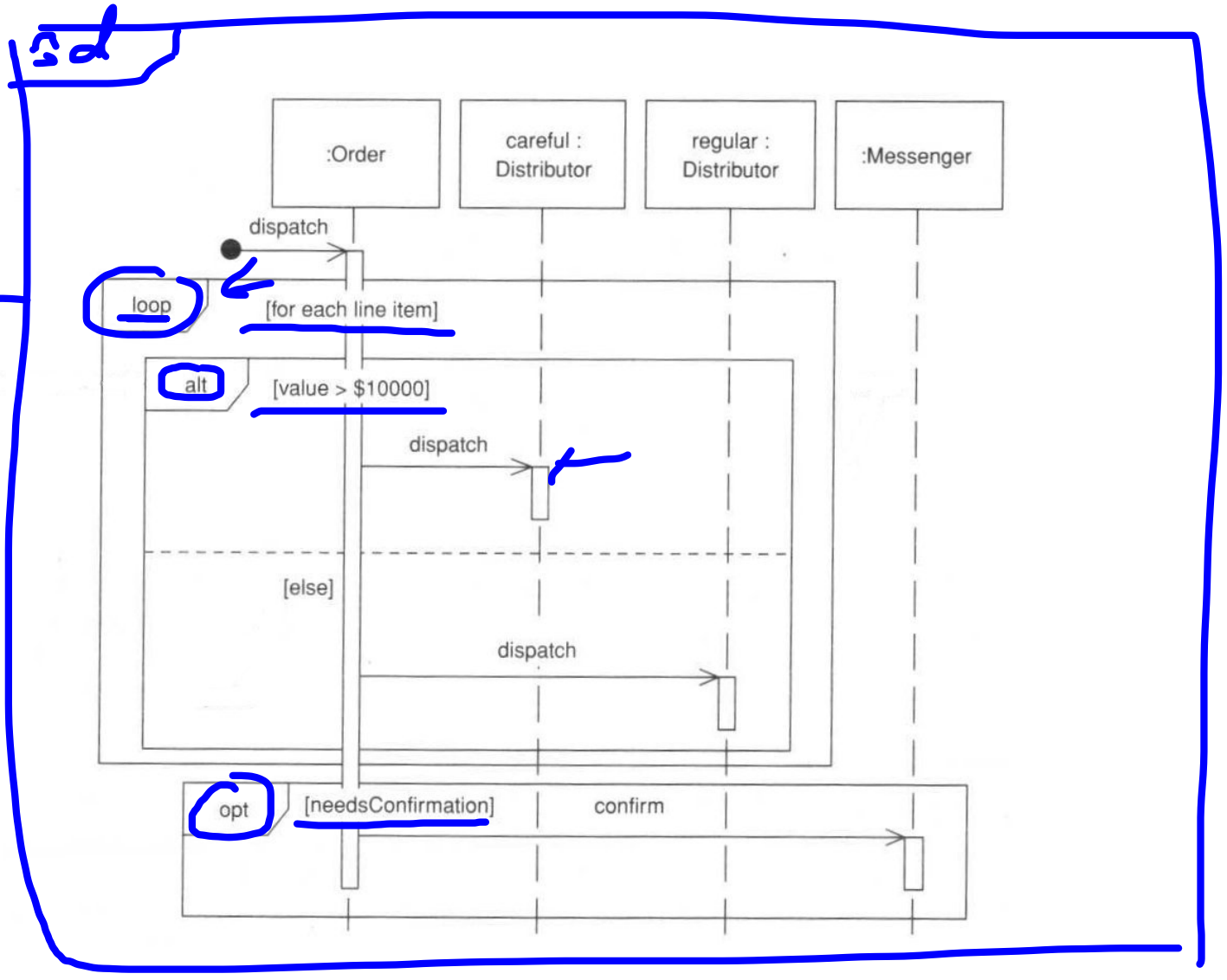
# Contents

# Interaction Frames Example

Realising an algorithm using a sequence diagram

```
public void dispatch() {
  for (LineItem lineItem : lineItems) {
    if (lineItem.getValue() > 10000) {
      careful.dispatch();
    } else {
      regular.dispatch();
    }
  }
  if (needsConfirmation()) {
    messenger.confirm();
  }
}
```

# Realisation with Interaction Frames



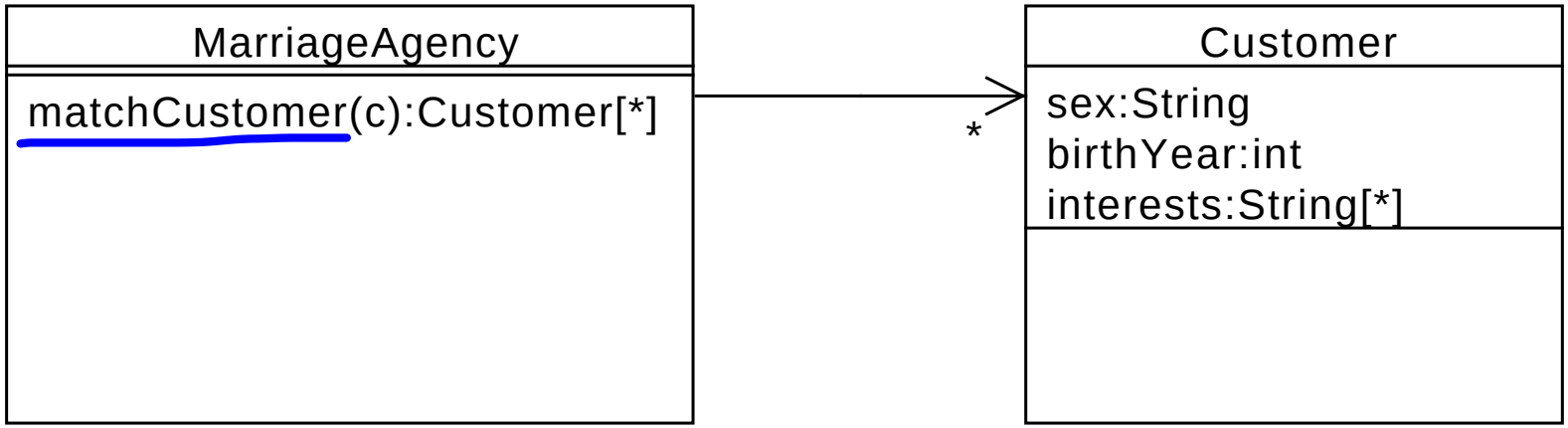

interaction frame

# Interaction Frame Operators I

| Operator | Meaning |
|---|---|
| alt | Alternative multiple fragments; only the one whose condition is true will execute (Figure 4.4). |
| opt | Optional; the fragment executes only if the supplied condition is true. Equivalent to an alt with only one trace (Figure 4.4). |
| par | Parallel; each fragment is run in parallel. |
| loop | Loop; the fragment may execute multiple times, and the guard indicates the basis of iteration (Figure 4.4). |
| ~~region~~ critical | Critical region; the fragment can have only one thread executing it at once. |
| neg | Negative; the fragment shows an invalid interaction. |
| ref | Reference; refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value. |
| sd | Sequence diagram; used to surround an entire sequence diagram, if you wish. |

# Contents

# Marriage Agency: centralized control

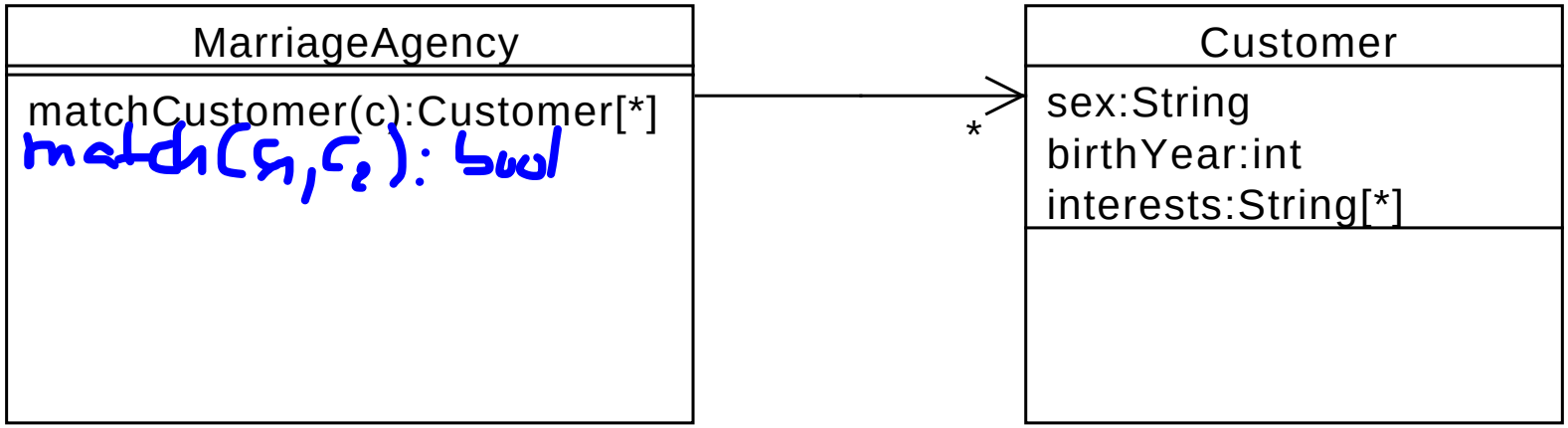



```
public class MarriageAgency{
   private List<Customer> customers = new ArrayList<Customer>();
   public List<Customer> matchCustomer(c) {
      List<Customer> matches = new ArrayList<Customer>();
      for (Customer candidate : customers) {
         ... // if customer matches candidate add to variable matches
      }
      return candidate;
   }
}
```

matches

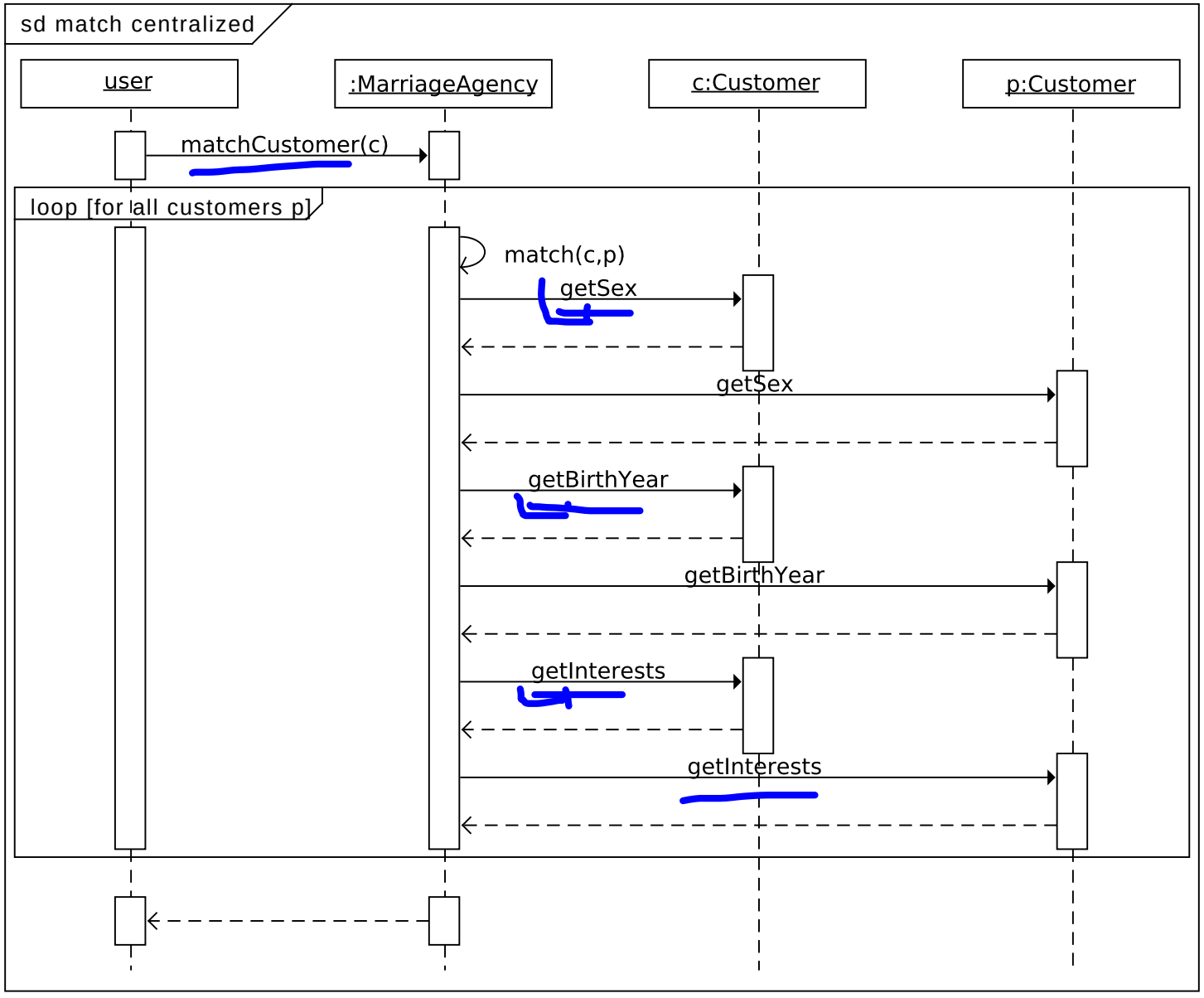
↳ if( cand.sex != c.sex &&
Math.abs( cand.by -
c.by) < 10) {

# Marriage Agency: centralized control

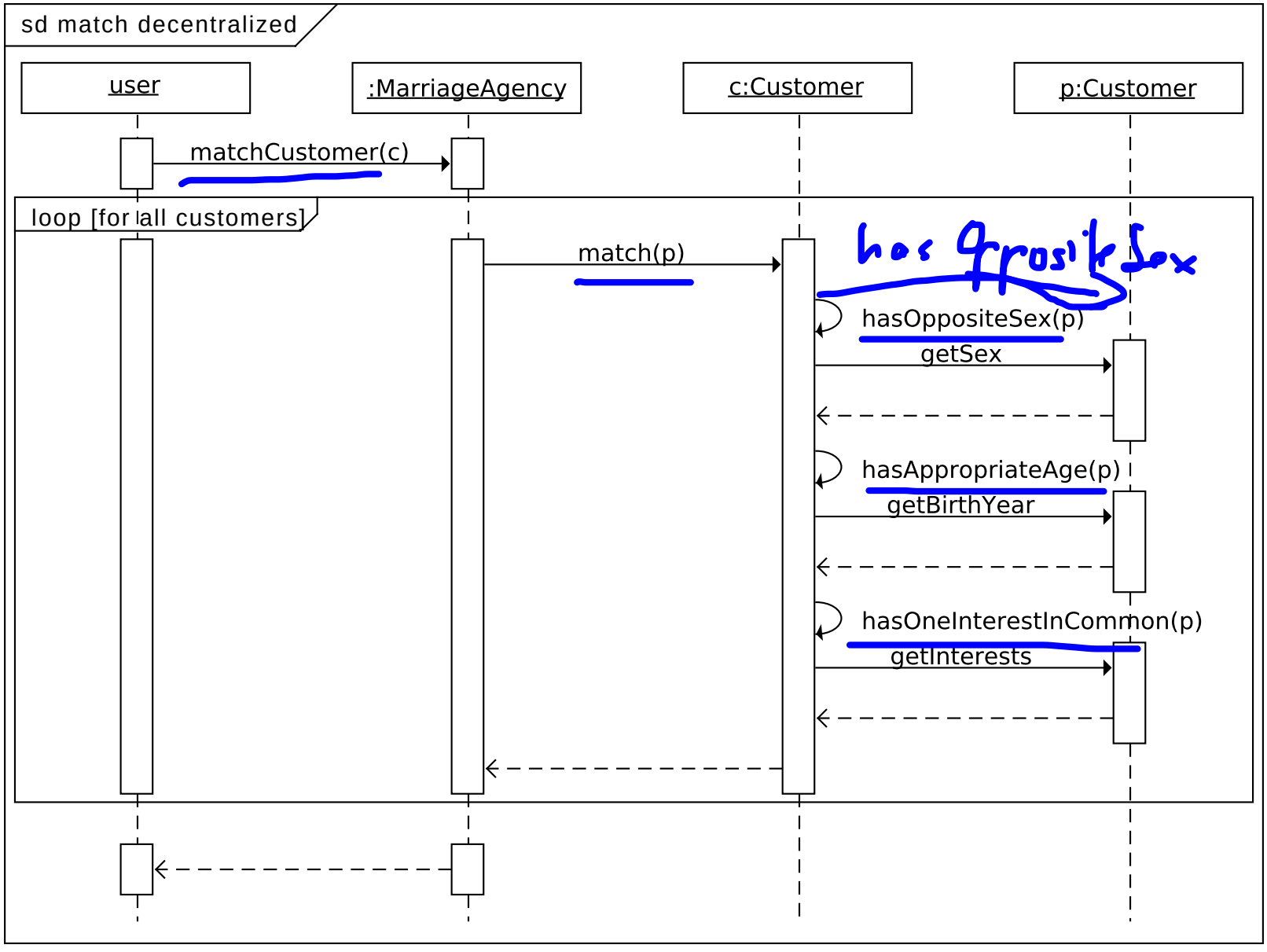



```
public class MarriageAgency{
   private List<Customer> customers = new ArrayList<Customer>();
   public List<Customer> matchCustomer(c) {
      List<Customer> matches = new ArrayList<Customer>();
      for (Customer candidate : customers) {
         // if customer matches candidate add to variable matches
         if (match(c,candidate)) {
            matches.add(candidate);
         }
      return candidate;
      }
   }
   public bool match(Customer customer, Customer candidate) {
      ....
   }
}
```

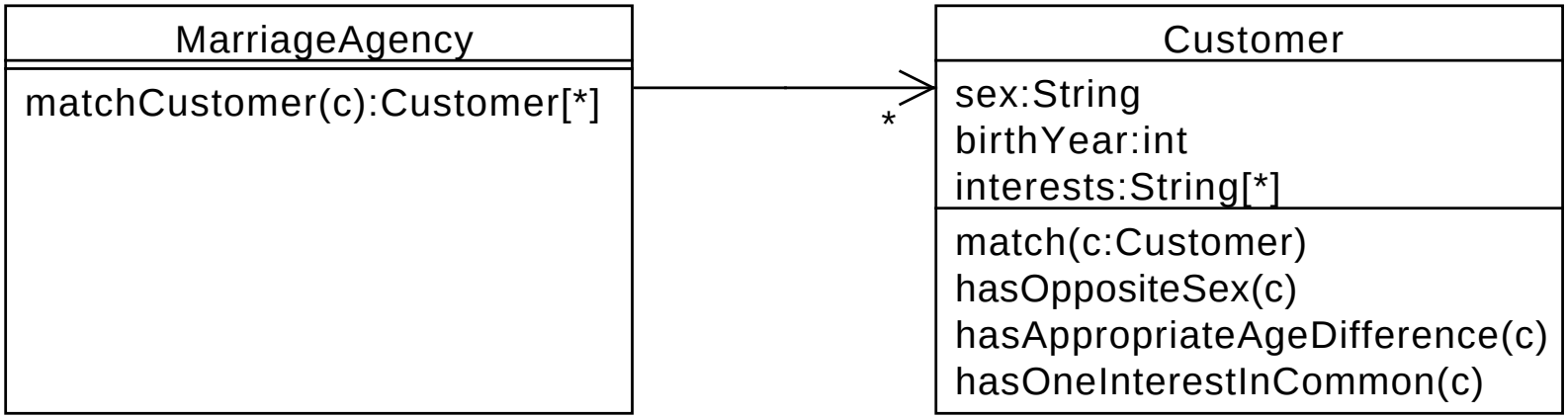
# Marriage Agency: centralized control



# Marriage Agency: decentralized/distributed control

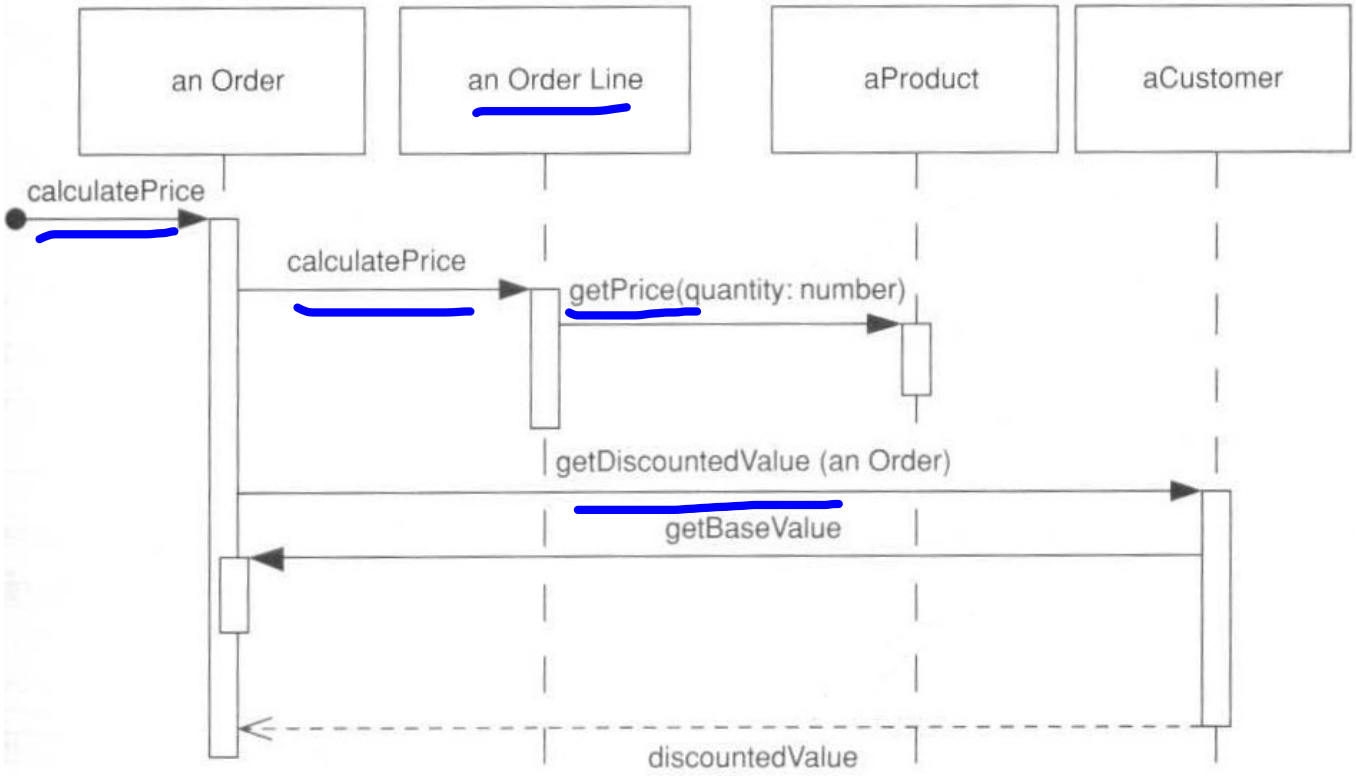


# Marriage Agency: decentralized/distributed control

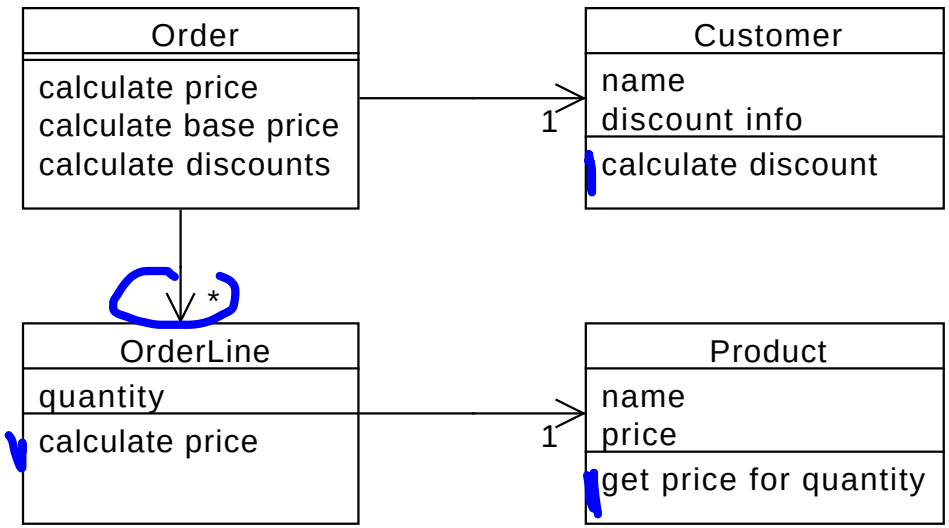



```
public class MarriageAgency{
   private List<Customer> customers = new ArrayList<Customer>();
   public List<Customer> matchCustomer(c) {
      List<Customer> matches = new ArrayList<Customer>();
      for (Customer candidate : customers) {
         if (c.match(candidate)) {
            matches.add(candidate);
         }
      return candidate;
      }
   }
}
public class Customer {
   ...
   public bool match(Customer candidate) {
      ....
   }
}
```

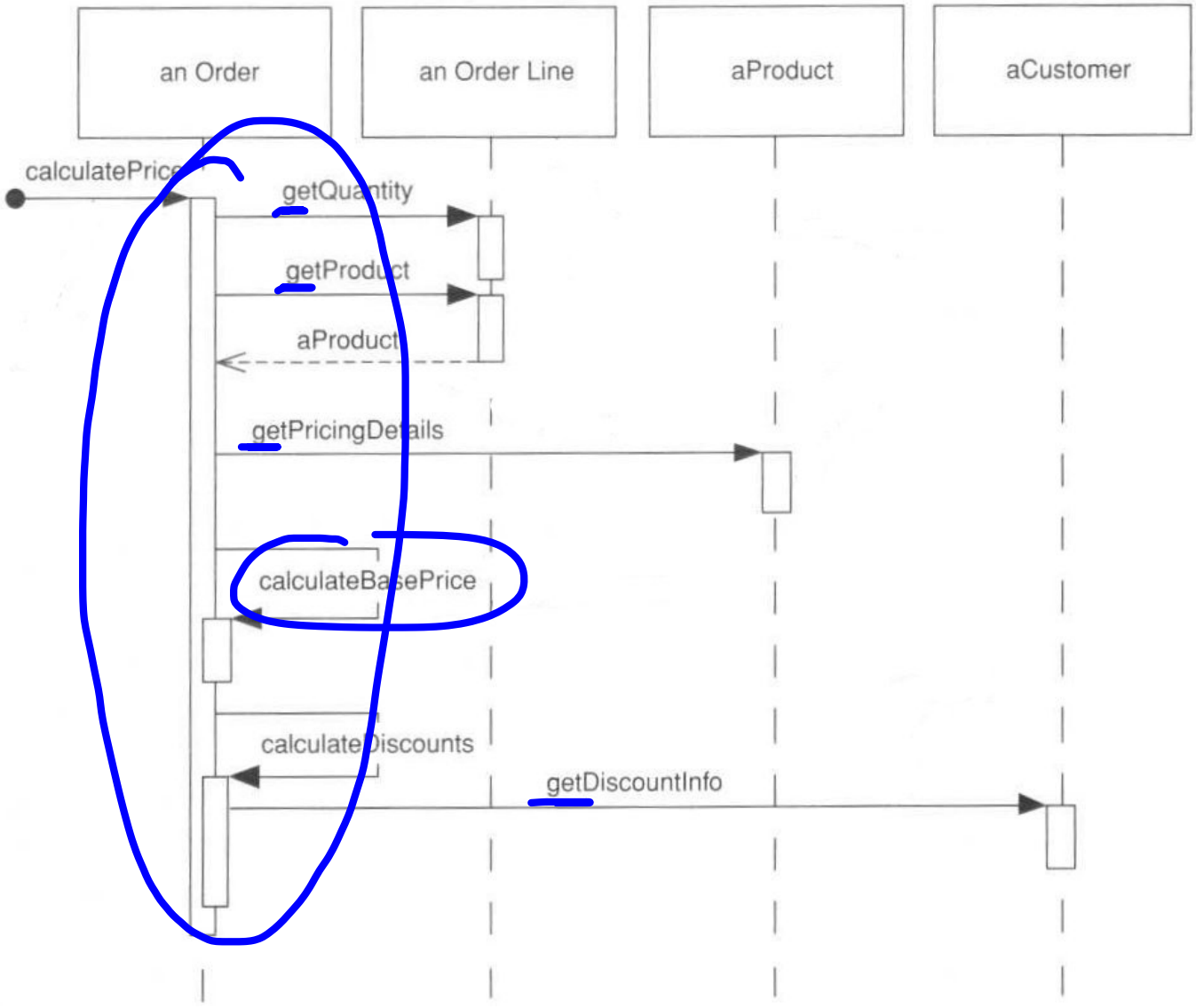
# Distributed control



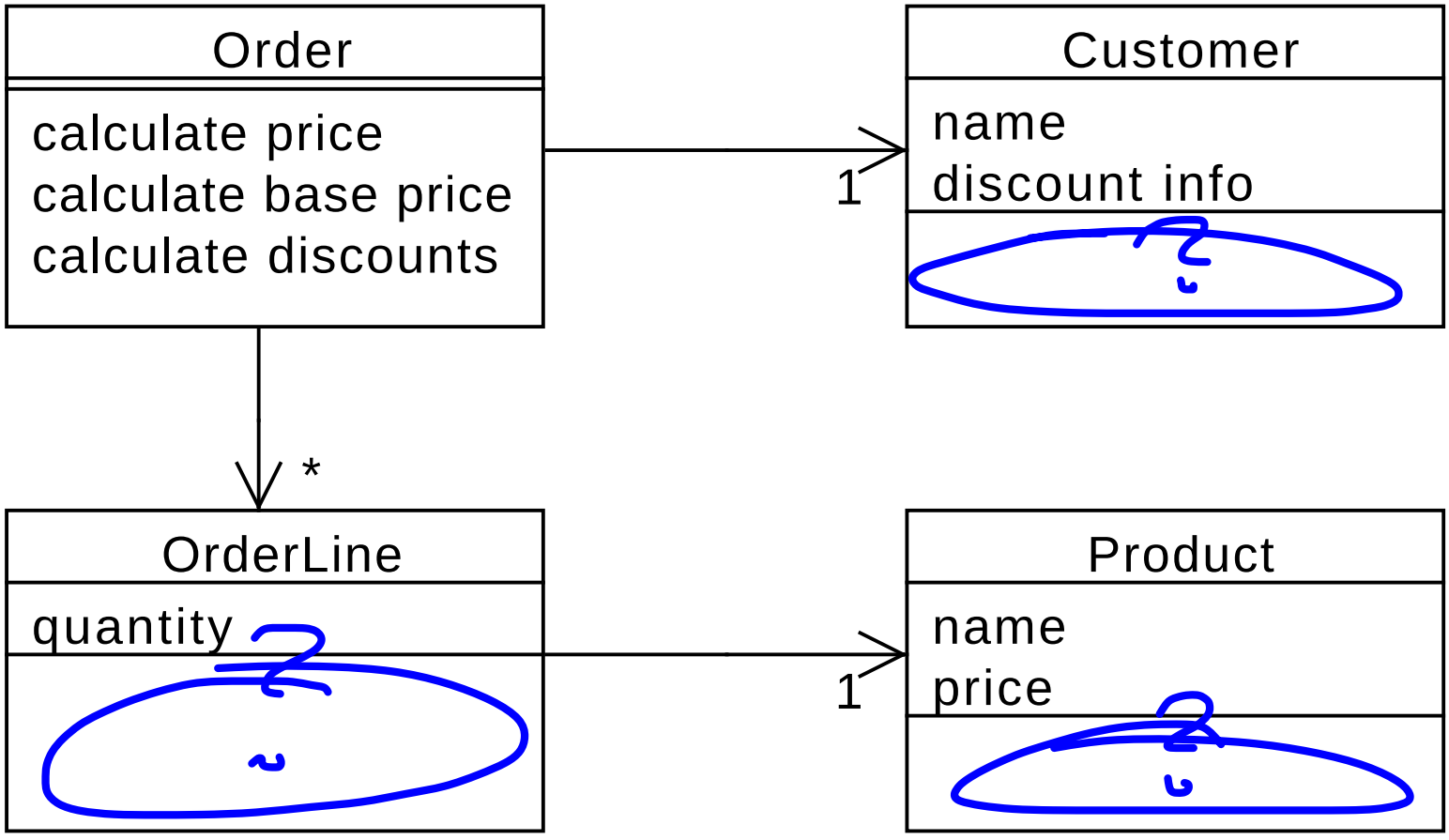
# Distributed Control: Class diagram



# Centralised control



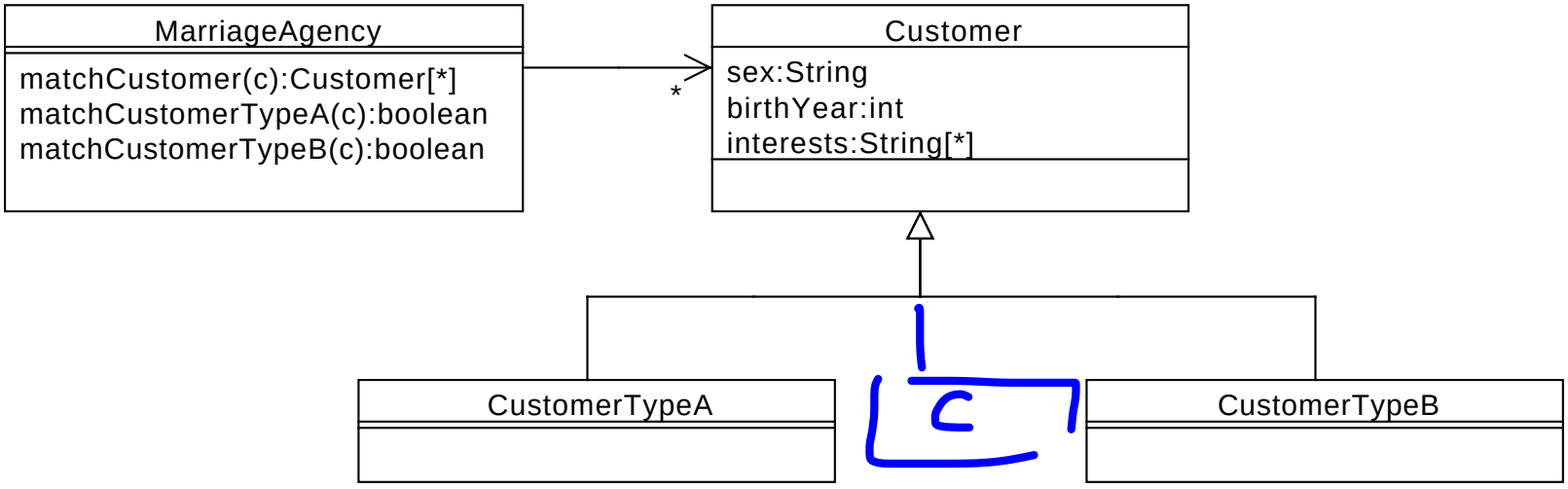
# Centralized control



# Centralized vs Distributed control

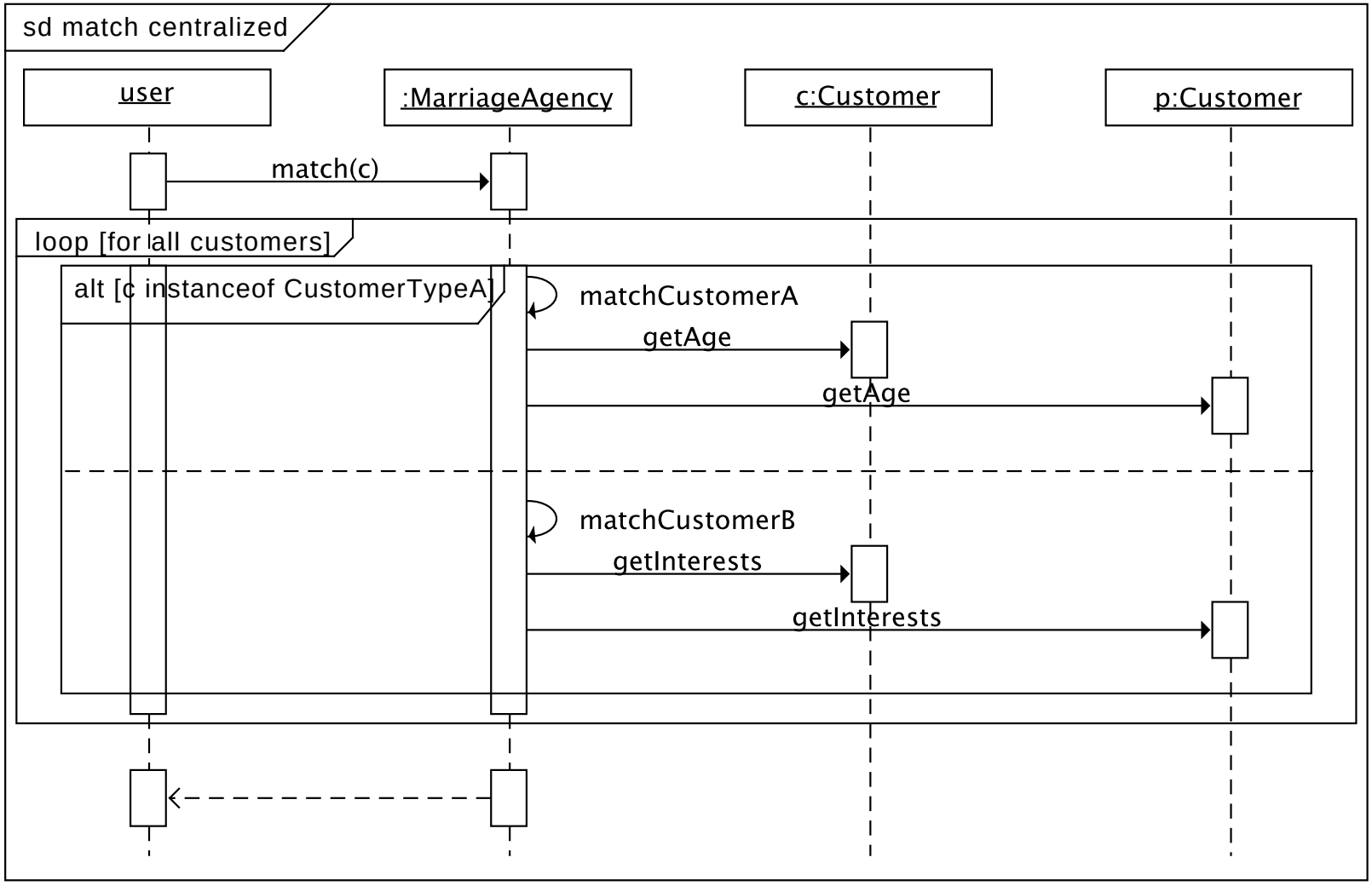
- Centralized control
  - **One method**
  - Data objects
  - → procedural programming language
- Distributed control
  - Objects **collaborate**
  - Objects = data *and* behaviour
  - → Object-orientation
- Advantage
  - Easy to adapt
  - → Design for *change*

# Design for change: centralized control

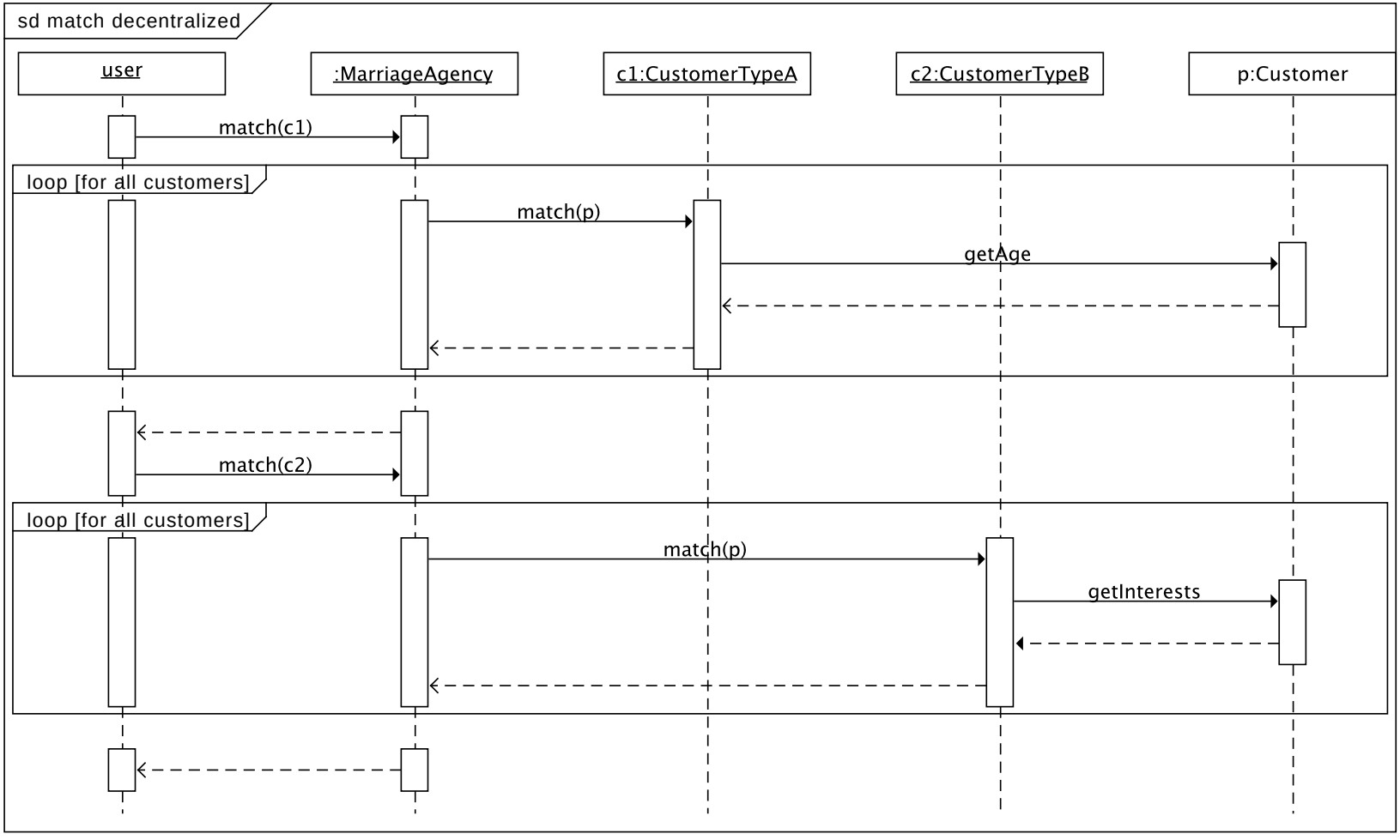



```
public List<Customer> matchCustomer(Customer c) {
   List<Customer> r = new ArrayList<Customer>();
   for (Customer p : customers) {
      if (c instanceof CustomerA) {
        if (matchCustomerTypeA(p)) { r.add(p); }
        continue;
      }
      if (c instanceof CustomerB) {
        if (matchCustomerTypeB(p)) { r.add(p); }
        continue;
      }
   }
   return r;
}
```

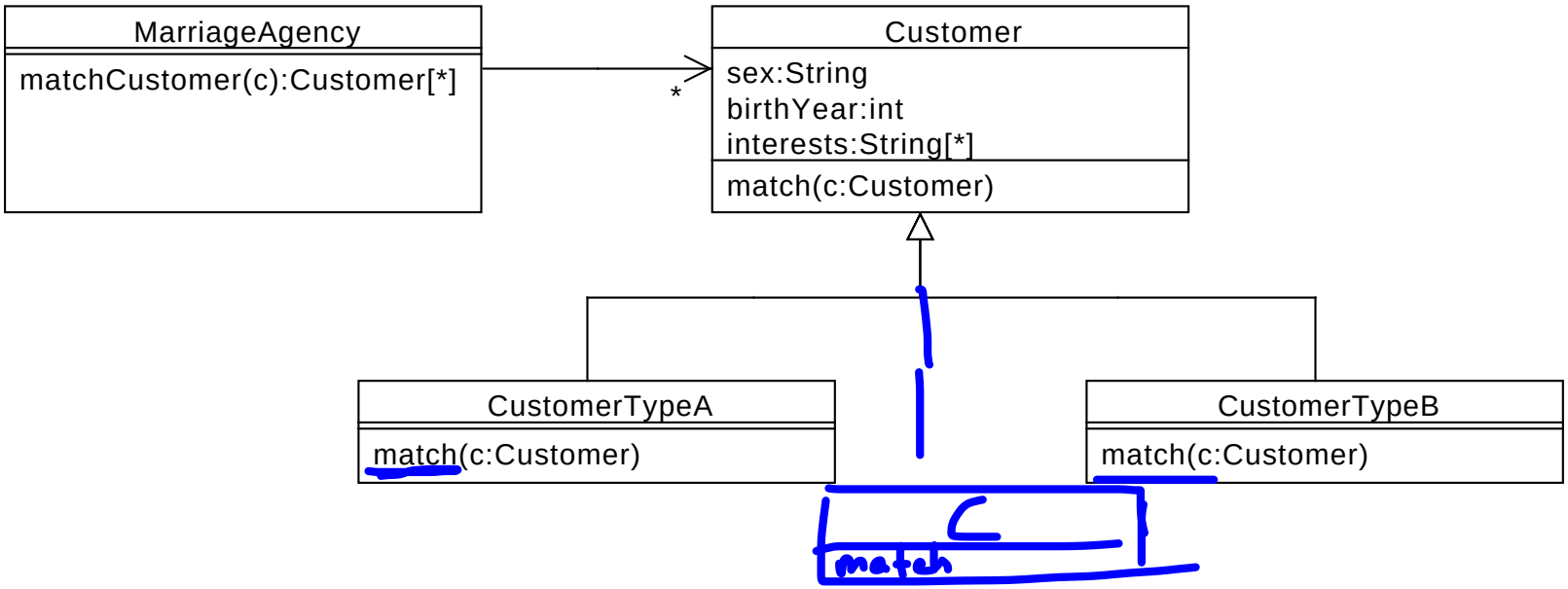
# Design for change: centralized control



# Design for change: decentralized control



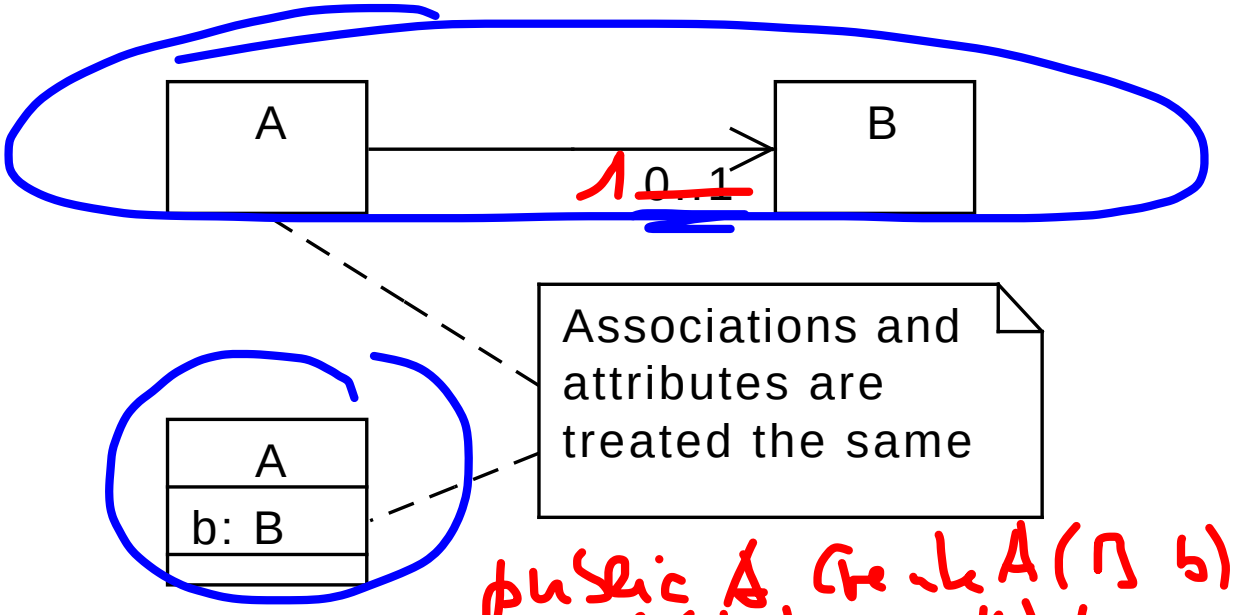
# Design for change: decentralized control




```
public List<Customer> matchCustomer(Customer c) {
   List<Customer> matches = new ArrayList<Customer>();
   for (Customer candidate : customers) {
      if (c.match(candidate)) { matches.add(p); }
   }
   return matches;
}
```

# Contents

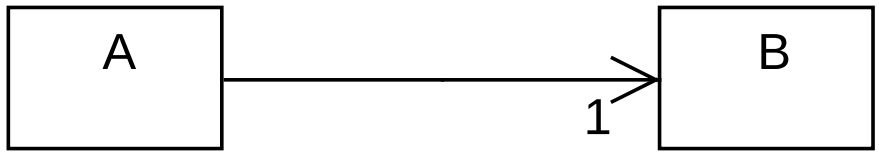
# Implementing Associations: Cardinality 0..1




- Field can be null

```
public class A {

    private B b;

    public B getB() {
       return b;
    }

    public void setB(B b) { this.b = b; }
}
```

# Implementing Associations: Cardinality 1

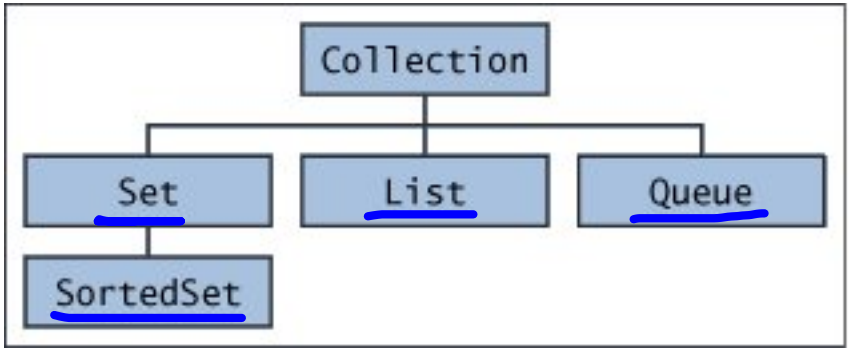



- Field may not be null

```
public class A {

   private B b = new B(); // 1st way of doing it

   public A(B b) { this.b = b;} // 2nd way

   public B getB() {                // 3rd way
     if (b == null) {b = computeB();}
     return b;
   }

   public void setB(B b) { if (b != null) {this.b = b;} }
}
```

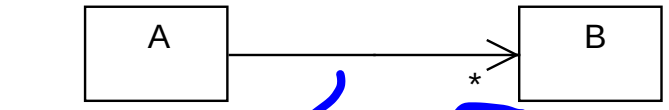
# Interface *Collection<E>*

| Operation | Description |
|---|---|
| `boolean add(E e)` | returns **false** if e is in the collection |
| `boolean remove(E e)` | returns **true** if e is in the collection |
| `boolean contains(E e)` | returns **true** if e is in the collection |
| `Iterator<E> iterator()` | allows to iterate over the collection |
| `int size()` | number of elements |



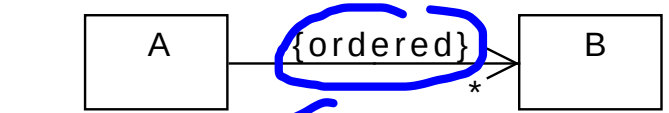

Interfaces

# Implementing Associations: Cardinality *



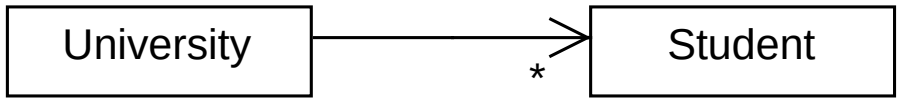

Default: Unordered, no duplicates

```
public class A {
   private Set<B> bs = new HashSet<B>();
   ...
}
```




```
public class A {
   private List<B> bs = new ArrayList<B>();
   ...
}
```

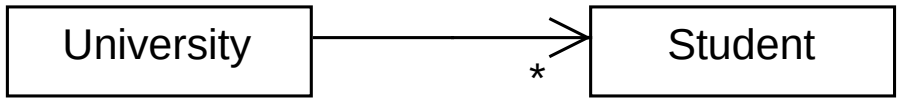
# Encapsulation problem: getStudents




```
University dtu = new University("DTU");
..
Set<Student> students = dtu.getStudents();
```

students.add(new Student("Peter"));

# Encapsulation problem: getStudents

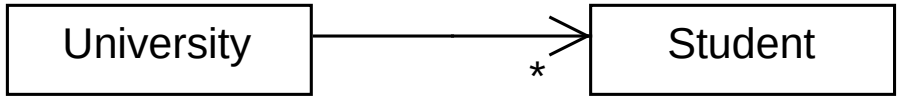



```
University dtu = new University("DTU");
..
Set<Student> students = dtu.getStudents();


Student hans = new Student("Hans");
students.add(hans);
Student ole = dtu.findStudentNamed("Ole");
students.remove(ole);
...
```

Solution: getStudents returns an unmodifiable set

```
public void Set<Student> getStudents() {
   return Collections.unmodifiableSet(students);
}
```

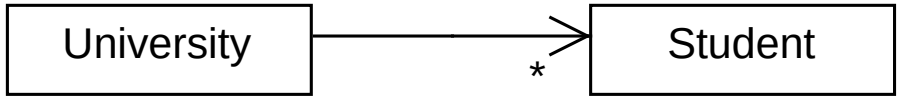
# Encapsulation problem: setStudents




```
University dtu = new University("DTU");
..
Set<Student> students = new HashSet<Student>();
dtu.setStudents(students);
```

student.add(new Student("Peter"))

# Encapsulation problem: setStudents

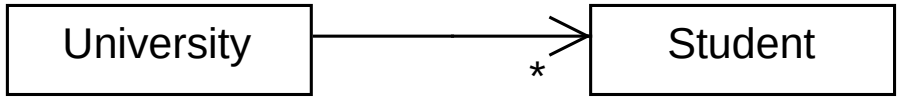



```
University dtu = new University("DTU");
..
Set<Student> students = new HashSet<Student>();
dtu.setStudents(students);


Student hans = new Student("Hans");
students.add(hans);
Student ole = dtu.findStudentNamed("Ole");
students.remove(ole);
...
```

## Solution: setStudents copies the set

```
   public void setStudents(Set<Student> stds) {
      students = new HashSet<Student>(stds);
}
```

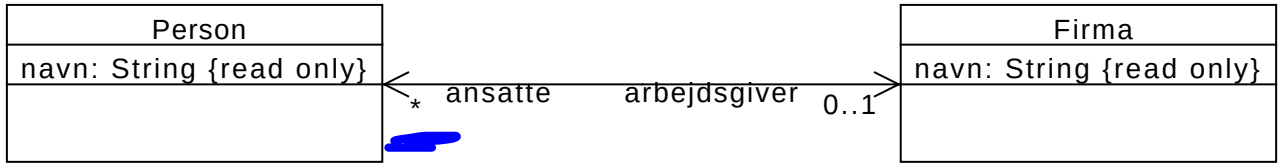
# Solution: How to change the association?



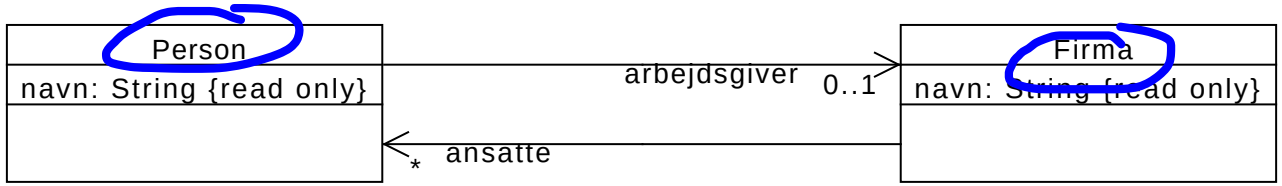

```
public class University {

   private Set<Student> bs = new HashSet<Student>();

   public void addStudent(Student s) {students.add(student);}
   public void containsStudent(Student s) {return students.contains(s)
   public void removeStudent(Student s) {students.remove(s);}
}
```

Even better: domain specific methods like *registerStudent*

# Bi-directional associations

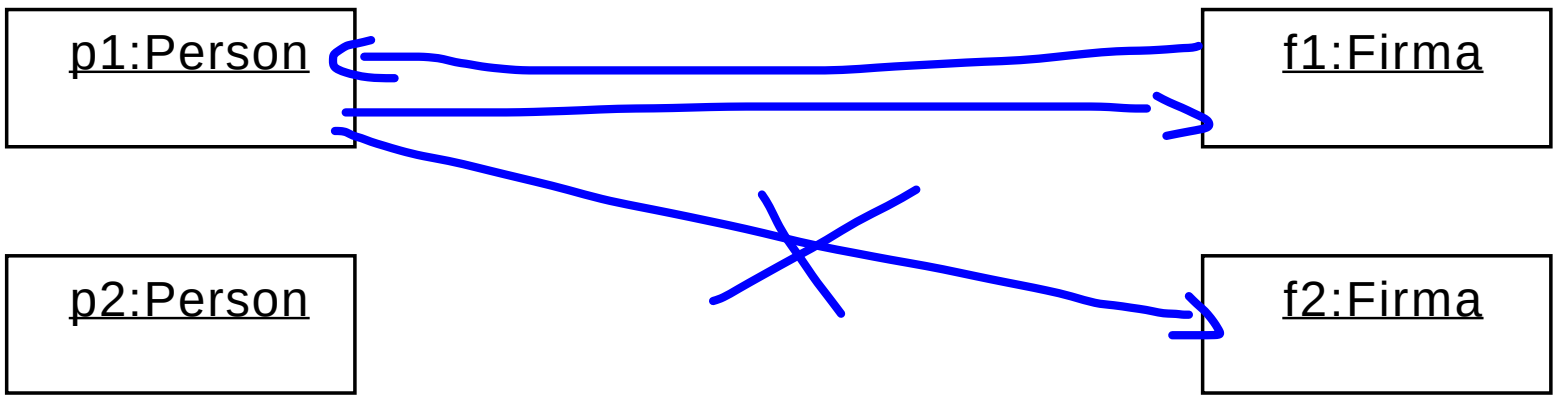



Implemented as two *uni-directional* associations

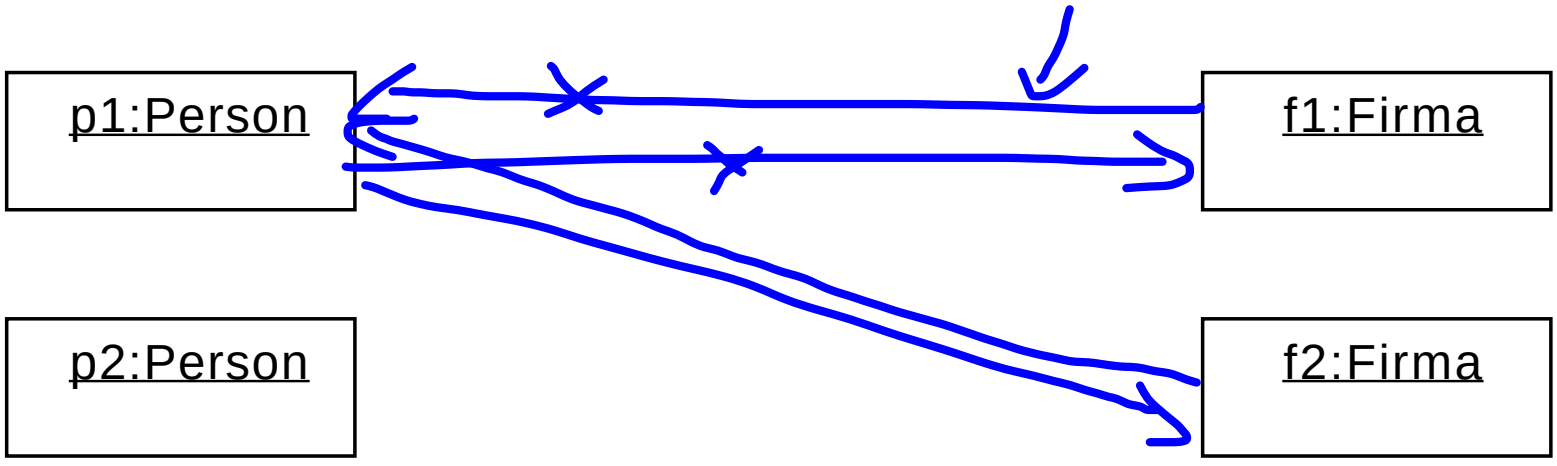



→ Problem of referential integrity

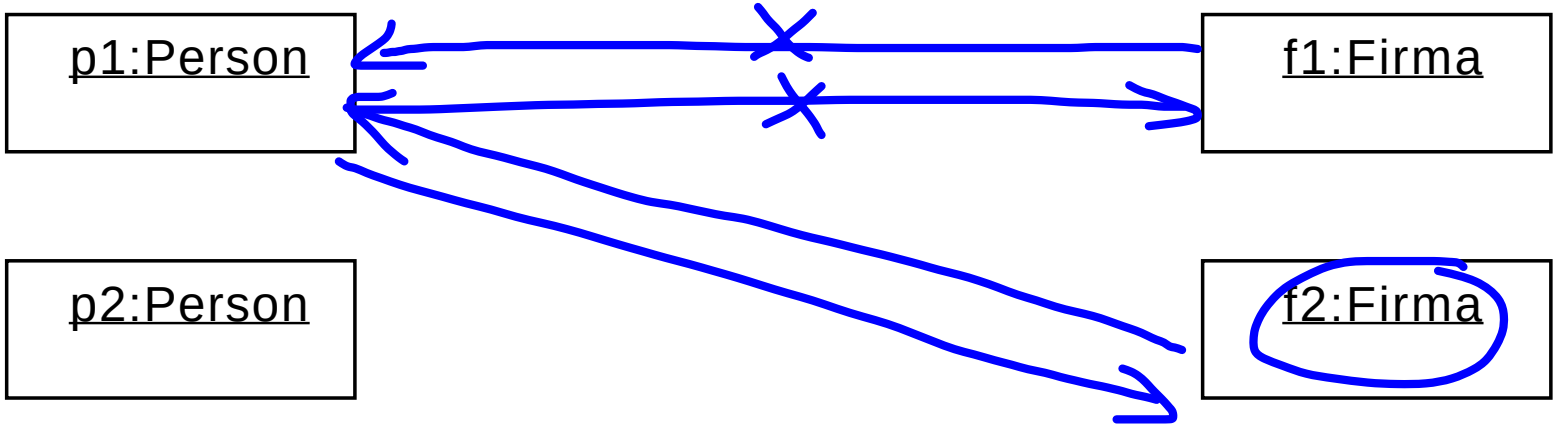
# Referential Integrity



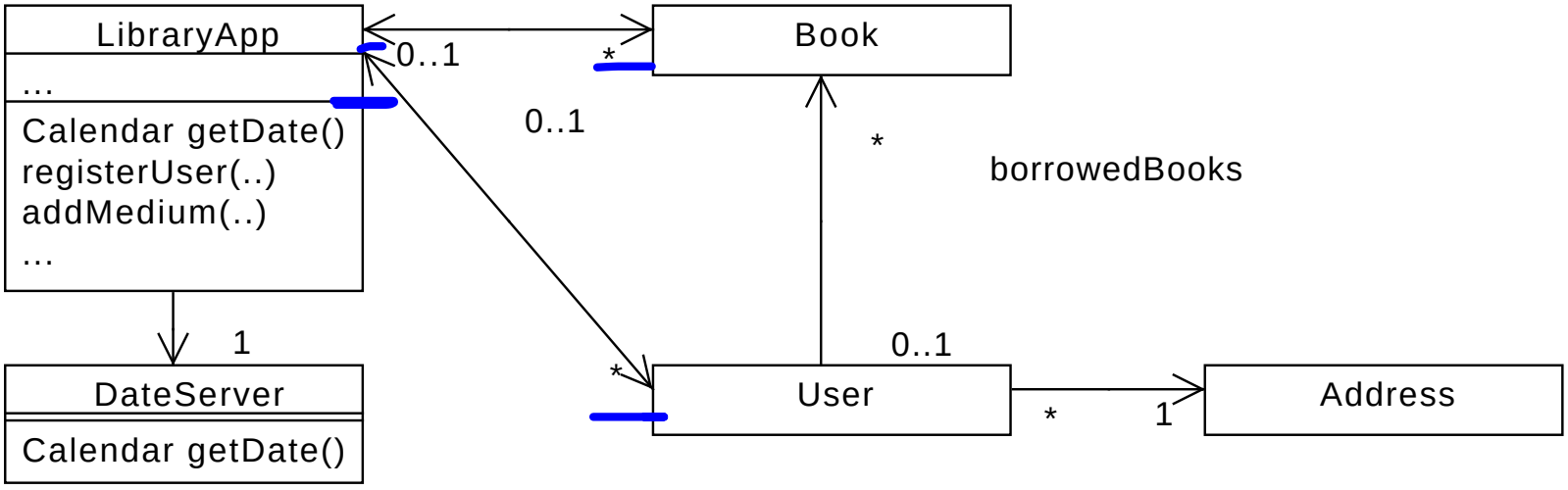
# Referential Integrity: setArbejdsgiver

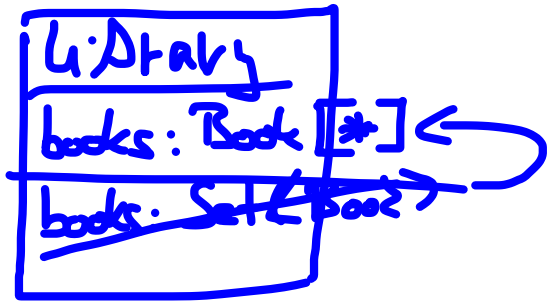


# Referential Integrity: addAnsatte



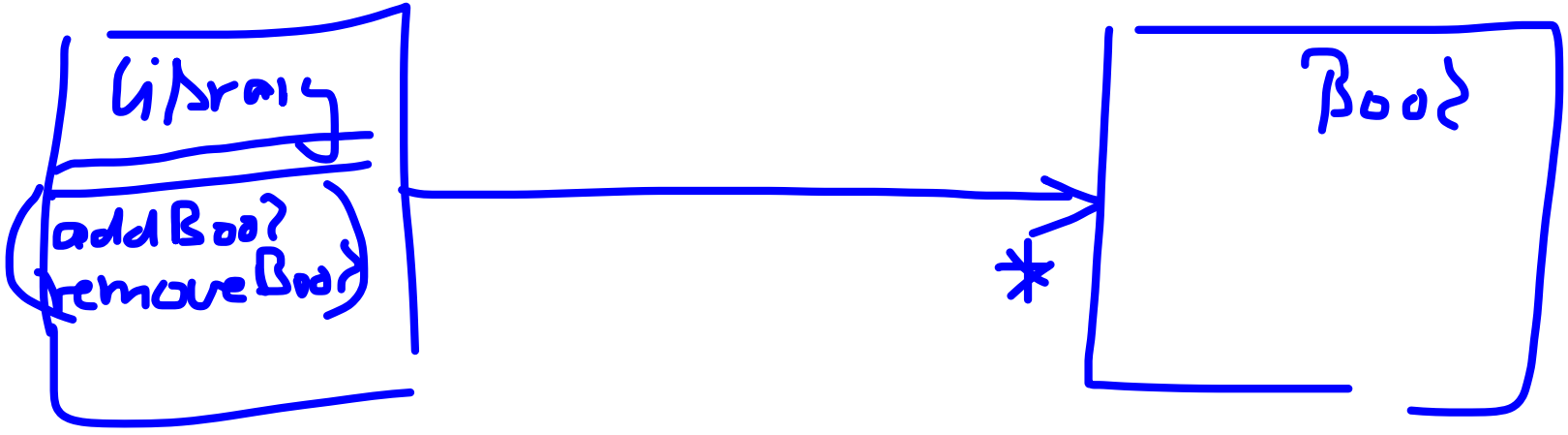
# Library application






What is the class diagram for the following code?

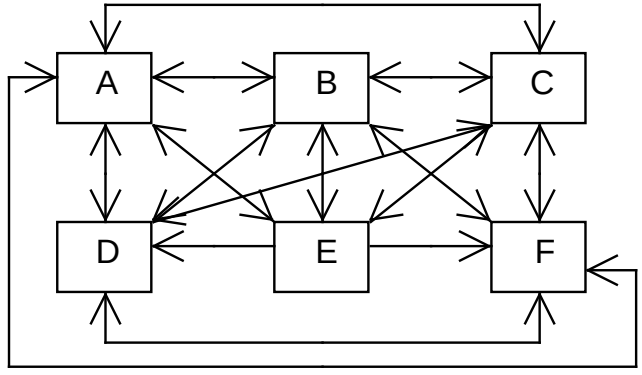
```
public class Library {
  Set<Book> books = new HashSet<Book>();
  public void addBook(Book book) { ... }
  public void removeBook(Book book) { ... }
  ....
}
public class Book { ... }
```

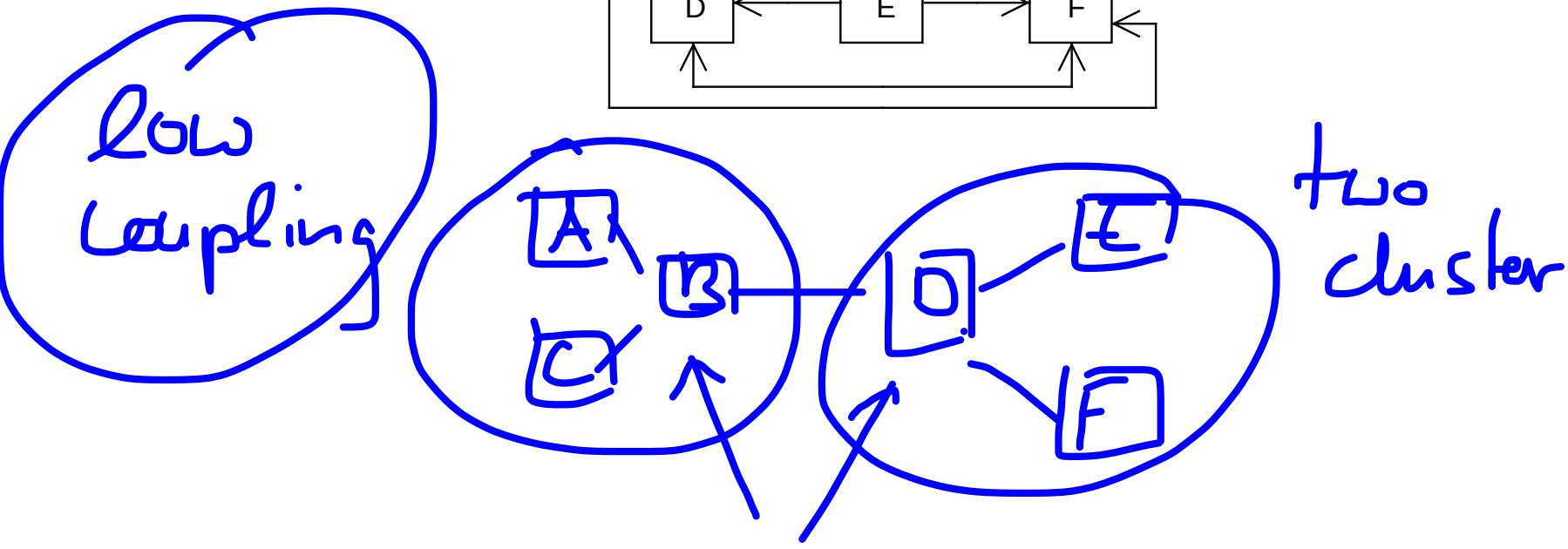


# Contents

# Low Coupling

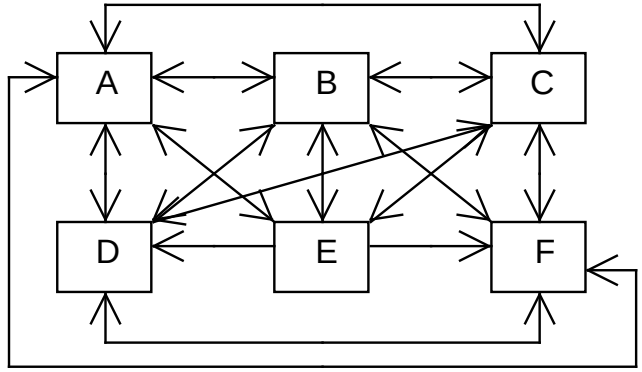
High coupling



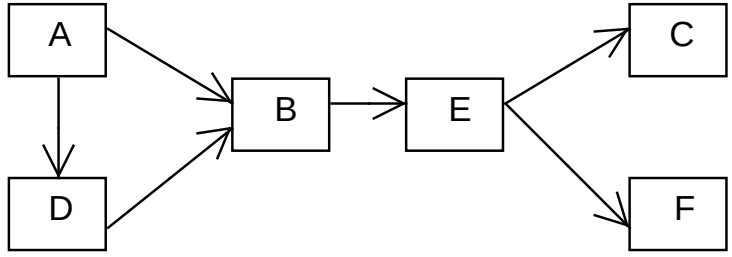



# Low Coupling

High coupling

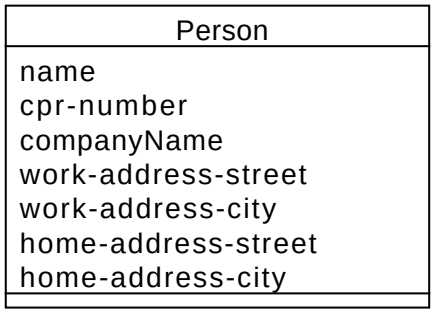



Low coupling



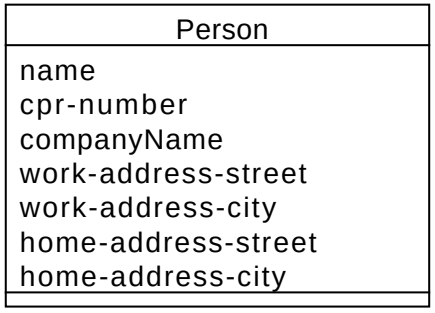
# High Cohesion

## Low Cohesion

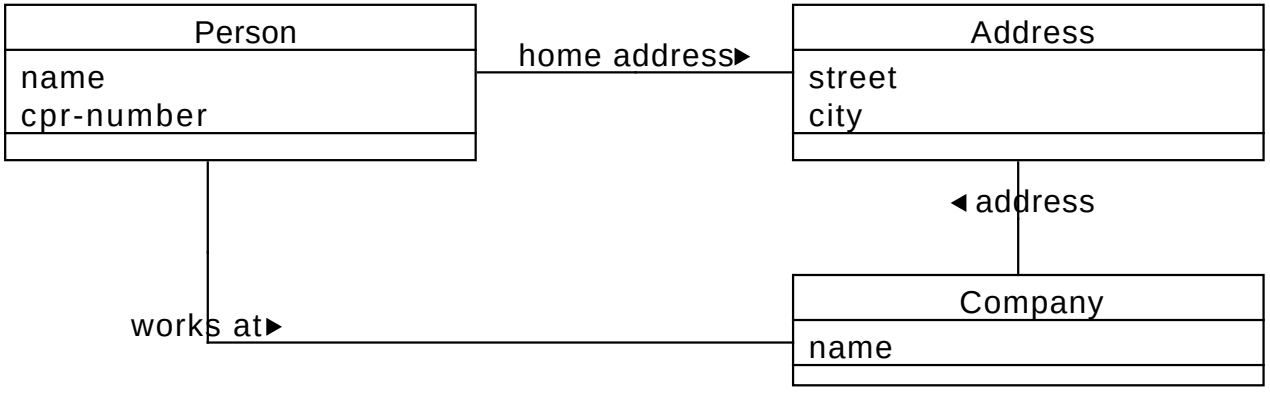


# High Cohesion

## Low Cohesion

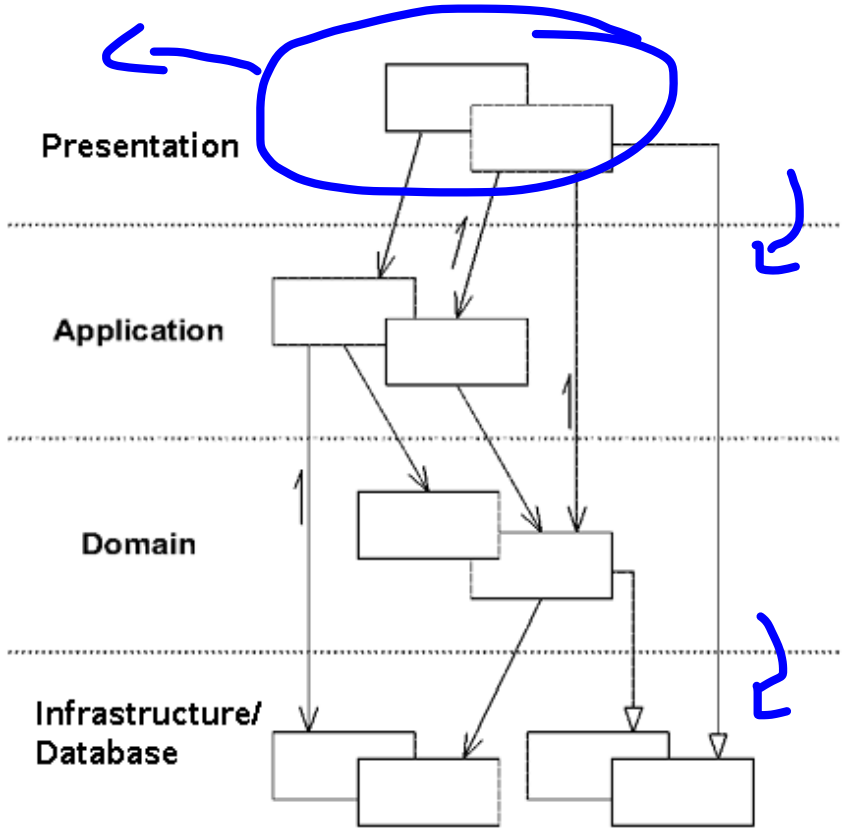



## High Cohesion



# Layered Architecture

high cohesion




low coupling

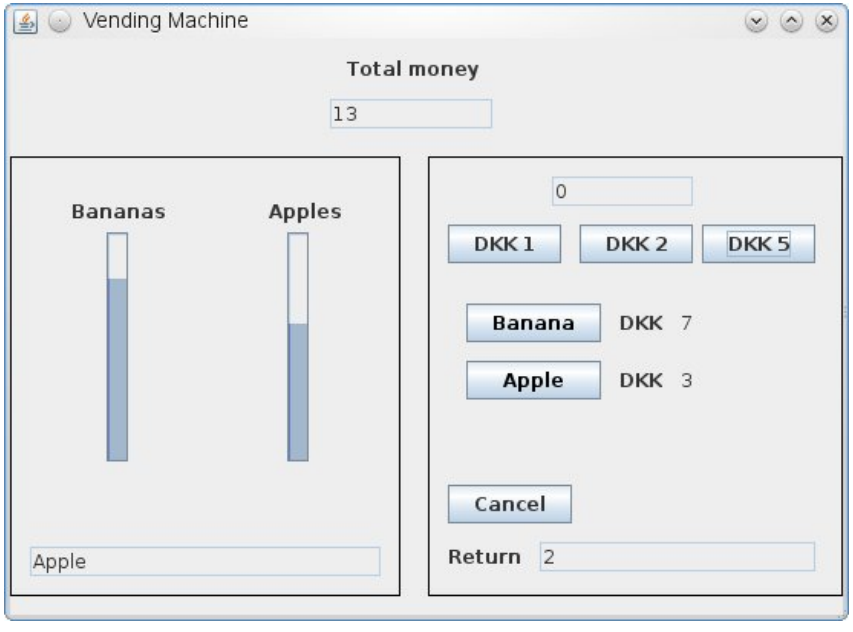
low coupling

Eric Evans, Domain Driven Design, Addison-Wesley, 2004

# Example Vending Machine

Two different presentation layers; same application layer

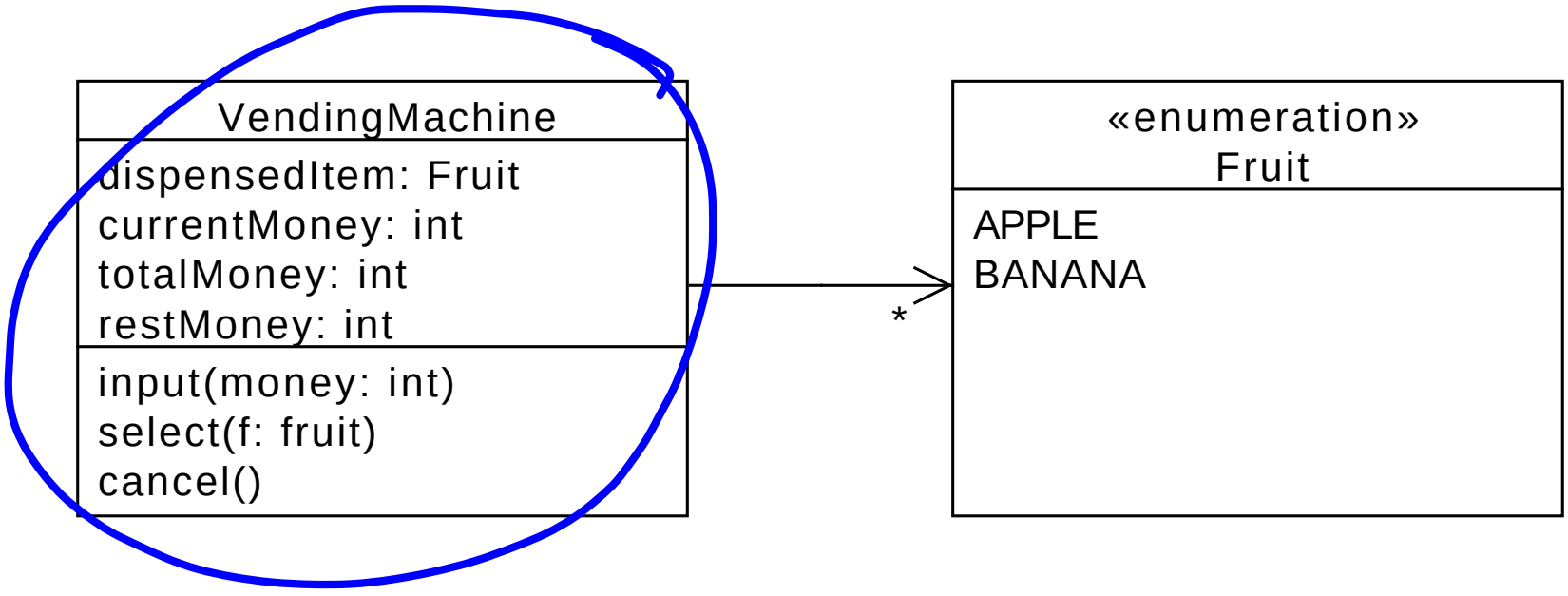
- Swing GUI



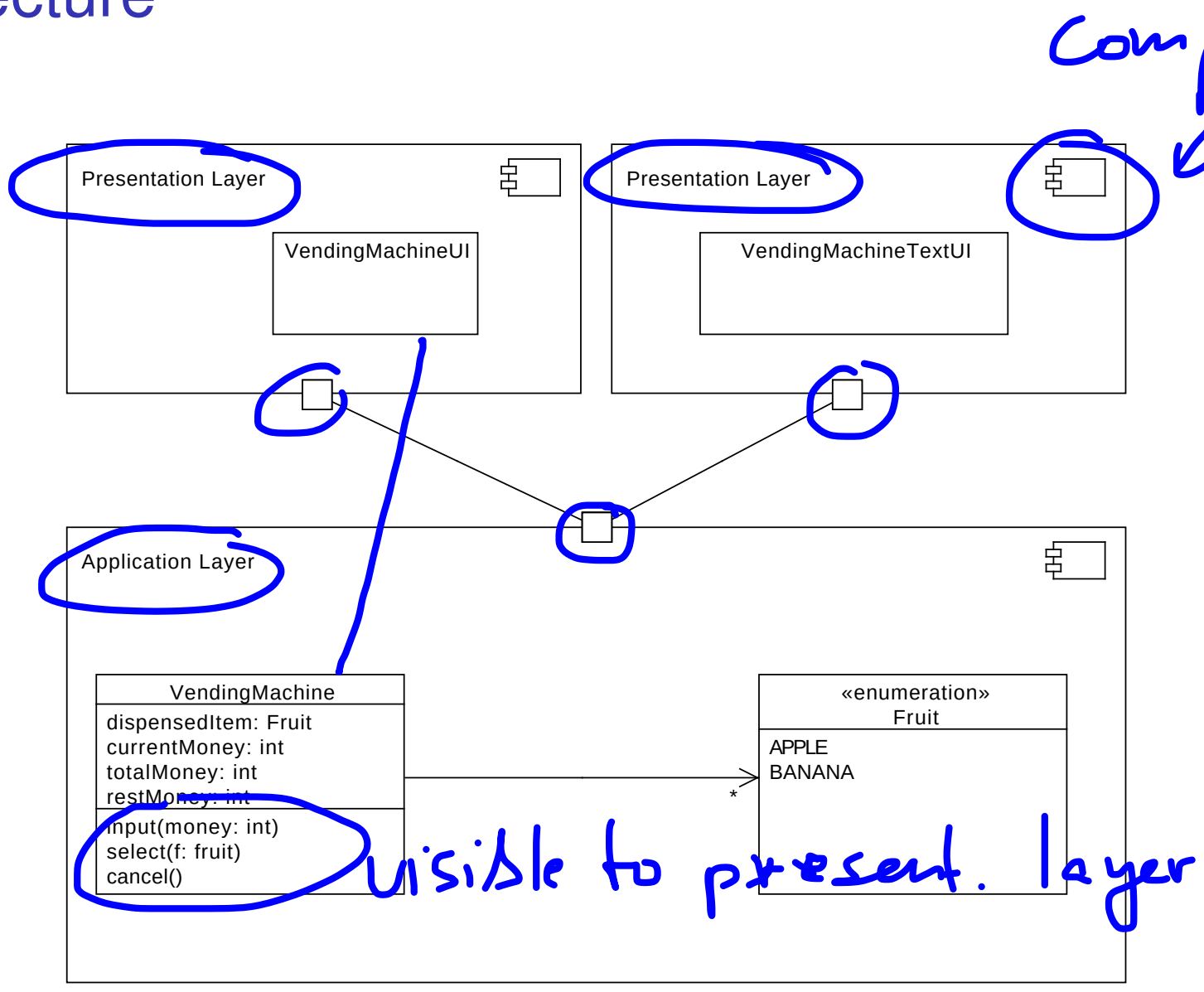

- Command line interface

```
Current Money: DKK 5
    0) Exit
    1) Input 1 DKK
    2) Input 2 DKK
    3) Input 5 DKK
    4) Select banana
    5) Select apple
    6) Cancel
Select a number (0-6):
Rest: DKK 2
Current Money: DKK 0
Dispensing: Apple
```

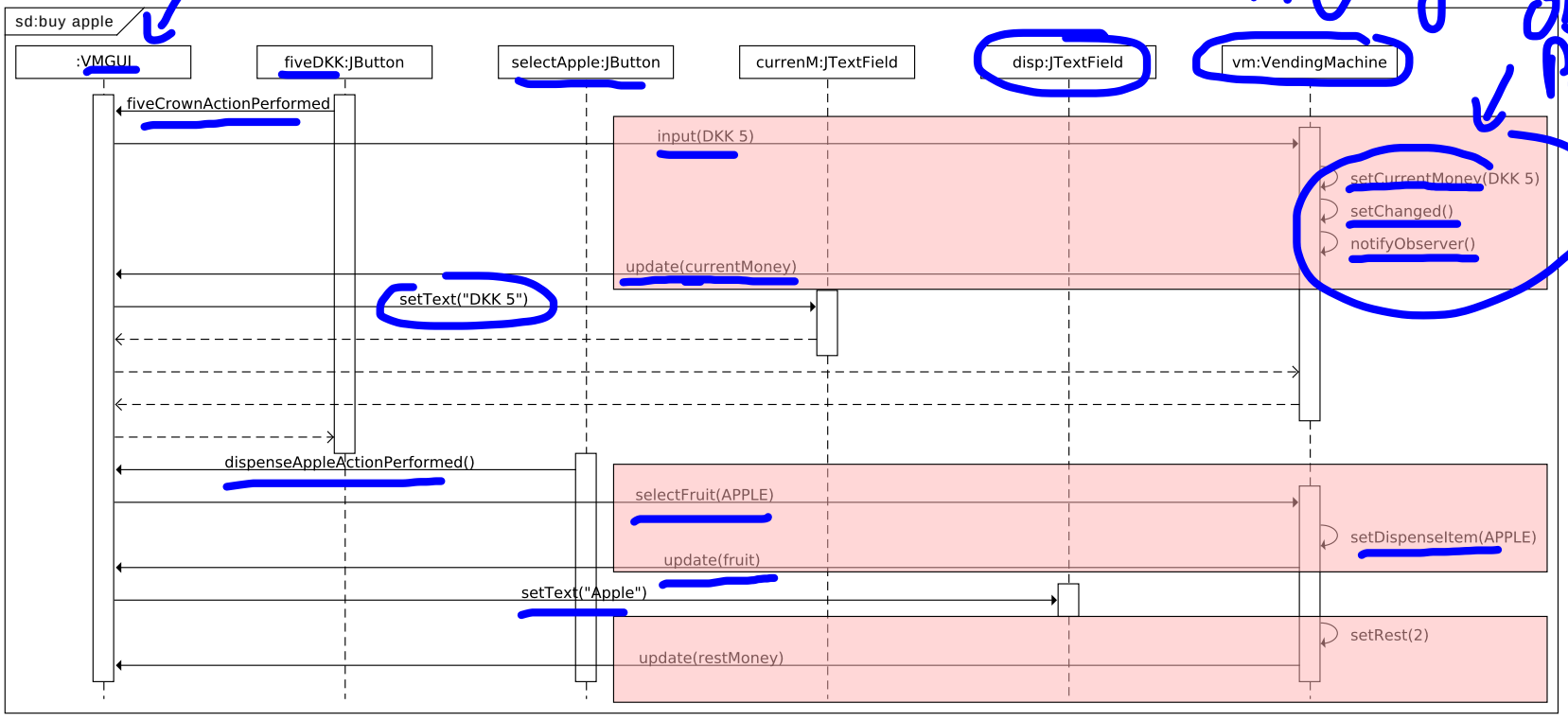
# Application Layer



# Architecture



# Presentation Layer: Swing GUI

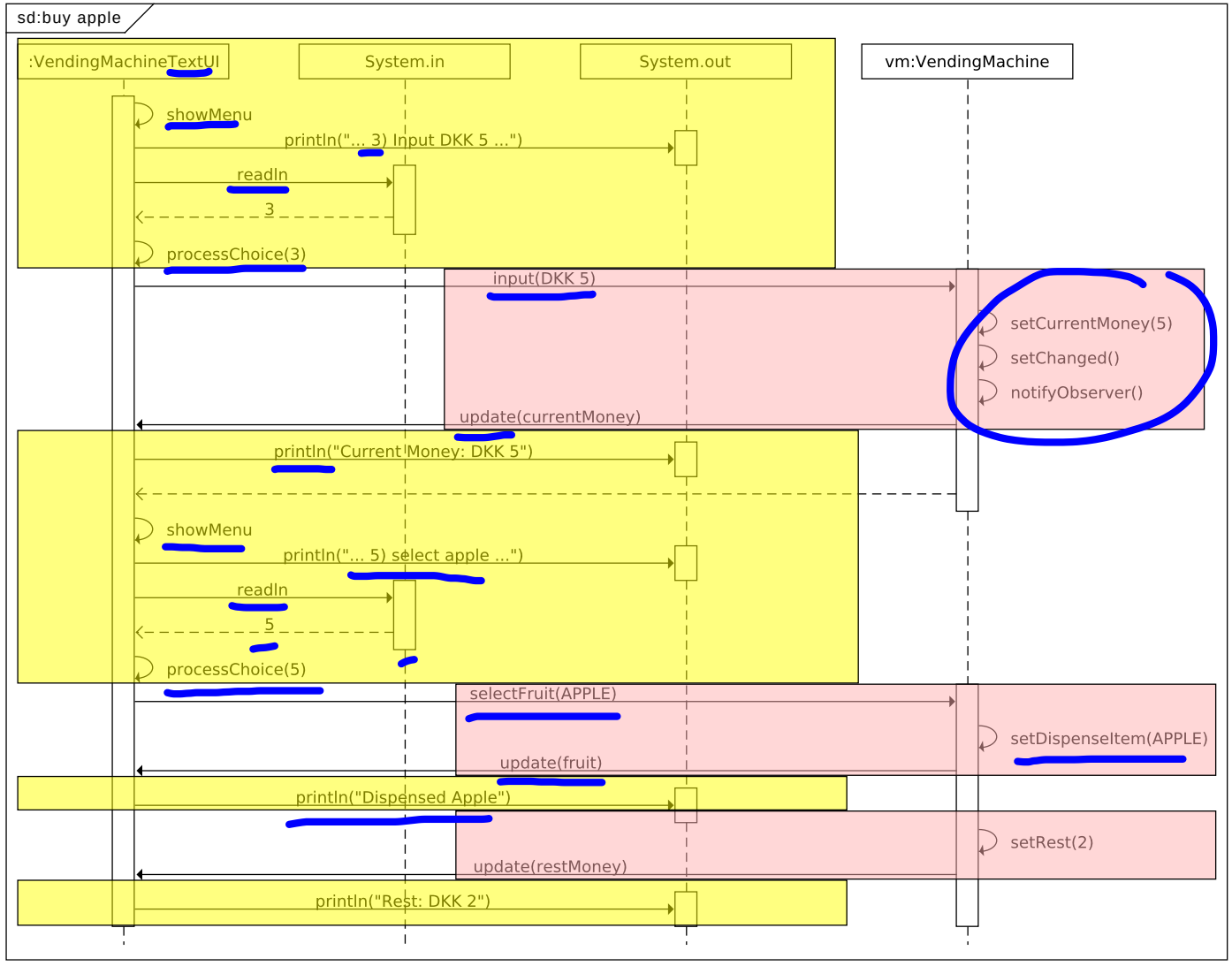


# Presentation Layer: Swing GUI

```
public class VendingMachineUI extends javax.swing.JFrame
      implements java.util.Observer {
   private VendingMachine vendingMachine = new VendingMachine(10, 10);
   ...
   private JButton fiveCrowns = new JButton();
   private JTextField currentM = new JTextField();
   ...

   private void initComponents() {
      fiveCrowns.setText("DKK 5");
      fiveCrowns.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
               vendingMachine.input(5);;
            }
         }
      )
      ...
   };
   ...
   public void update(Observable o, Object arg) {
      currentM.setText("" + vendingMachine.getCurrentMoney());
      ...
   }
}
```

↳ delegate to applic. layer

# Presentation Layer: Command Line Interface



# Presentation Layer: TextUI

```
public class VendingMachineTextUI implements Observer {
  VendingMachine vendingMachine;
  public static void main(String[] args) throws Exception {
    new VendingMachineTextUI(5,5).mainLoop(System.in, System.out);
  }
  public void mainLoop(InputStream in, PrintStream out) throws IOExcep
    BufferedReader rs = new BufferedReader(new InputStreamReader(in));
    do {
      showMenu(out);
      int number = Integer.valueOf(rs.readLine());
      processChoice(number, out);
    } while (number != 0);
  }
  private void processChoice(int number, PrintStream out) {
    switch (number) {
      case 3: vendingMachine.input(5); break;
      ...
    }
  }
  public void update(Observable o, Object arg) {
    NotificationType type = (NotificationType) arg;
    if (type == NotificationType.CURRENT_MONEY) {
      System.out.println("Current Money: DKK " +
        vendingMachine.getCurrentMoney());
      return;
    }
  }
}
```

delegates to the app. layer

# Advantages of the separation

1. Presentation layer easily changed
2. Additional presentation layers can be added easily without having to reimplement the business logic
   - mobile app in addition to desktop and Web application
3. Automatic tests

```
public void testInputDataSetA() {
    VendingMachine m = new VendingMachine(10, 10);
    m.input(1);
    m.input(2);
    assertEquals(3, m.getCurrentMoney());
    m.selectFruit(Fruit.APPLE);
    assertEquals(Fruit.APPLE, m.getDispensedItem());
}
```

- You should model the application and domain layer with your class diagrams and sequence diagrams